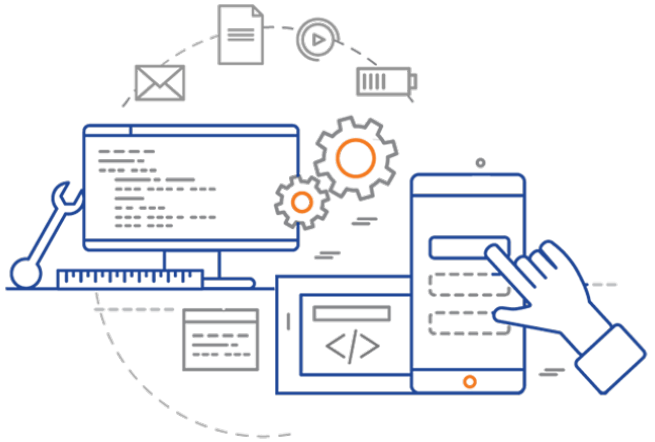


CS 398: Application Development

TDD & Unit Testing

Goals of testing; TDD; Unit Testing; JUnit & Kotlin

Why do we test?

The goal of testing is to ensure that the software that we produce meets our objectives when deployed into the environment in which it will be used, and when faced with real-world constraints.

Why do we test?

1. To gain confidence in the correctness of your results.

2. To gain confidence that you are handling edge cases and errors properly, which will result in a better user experience.

3. To produce an improved design, usually as a by-product of having written tests. The process of writing tests forces us to structure our code more carefully.

4. To help identify deficiencies and flaws in both software design and implementation.

How do we test?

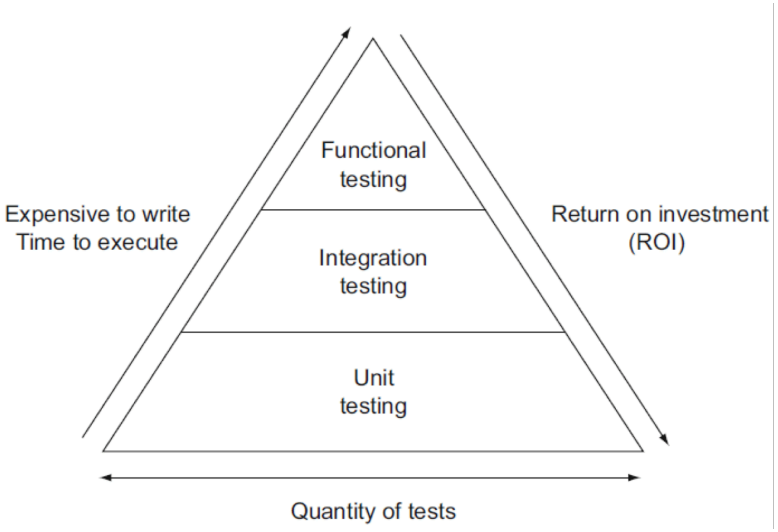
We produce automated tests that can confirm that our system is working as expected.

We can identify three types of tests:

1. **Unit tests**: tests operating at the class level (or smallest functional unit), which are meant to check the validity of low-level interfaces and systems.

2. **Integration tests**: testing across multiple classes or functional units, to check interaction between objects.

3. **System tests**: this is testing functionality from the perspective of the user; end-to-end feature testing. Sometimes called functional testing.

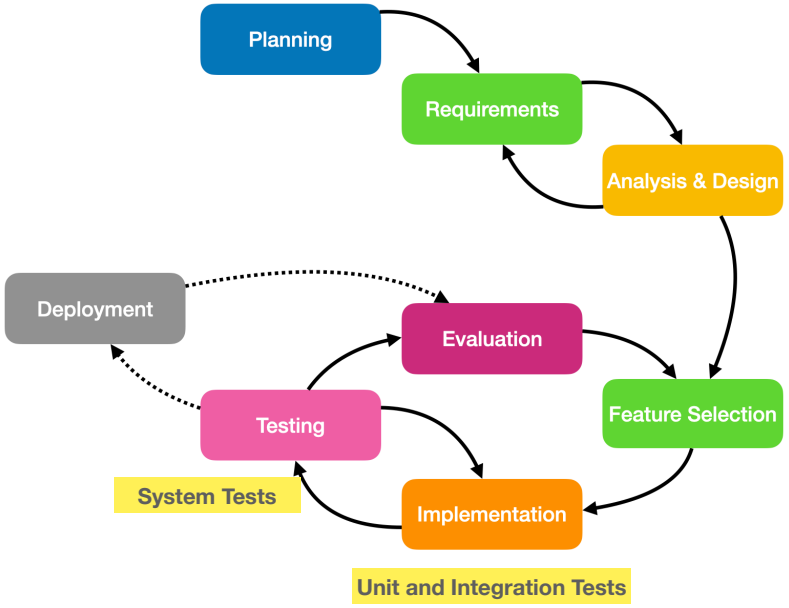


When should we test?

Traditional views were that testing should be done after implementation. This is costly. Testing is more useful when done *earlier* in the process.

Different tests are suitable for different parts of the development process:

- **Unit Tests**: done *during implementation*, when you are working on a class.
- **Integration Tests**: done *during implementation*, when you want to ensure that classes work together.
- **System Tests**: done when features are *complete and merged*, to ensure that the system continues working.



Test-Driven Development (TDD)

Test-Driven Development (TDD)

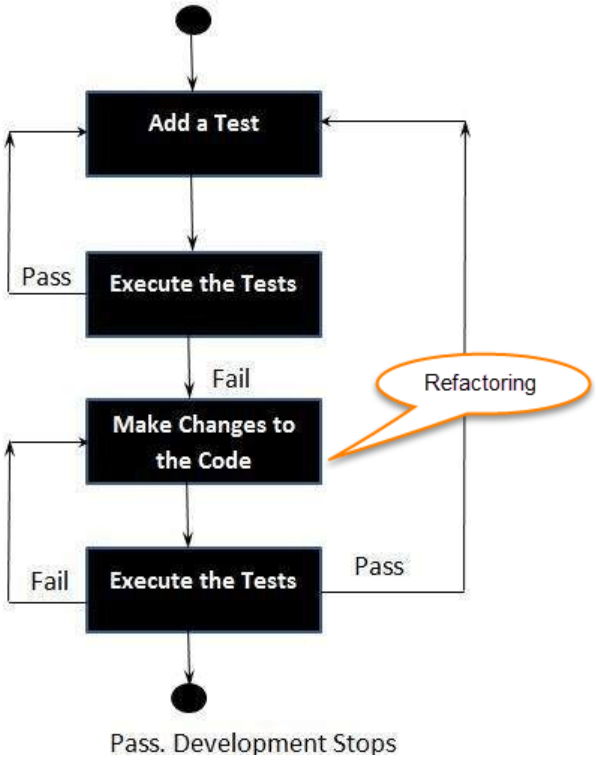
Promoted by Kent Beck around 2002 as an **Extreme Programming (XP) practice**.

The basic idea is that you write tests *before* writing the corresponding implementation code.

TDD development cycle

1. Define an interface or specification for your class or module.
2. Write a test against that interface.
3. Write the implementation code that causes the test to pass.
4. Repeat until completed.

TDD assumes 100% test coverage.



Why do we do TDD?

There are some clear benefits:

- **Early bug detection**. You are building up a set of tests that can verify that your code works as expected.
- **Better designs**. Making your code testable often means improving your interfaces, having clean separation of concerns, and cohesive classes. Testable code is by necessity better code.
- **Confidence to refactor**. You can make changes to your code and be confident that the tests will tell you if you have made a mistake.
- **Simplicity**. Code that is built up over time this way tends to be simpler to maintain and modify.

TDD is based on Unit and Integration Tests

Unit tests are meant to exercise the interface of a single class or module.

- Unit tests should be very quick to execute and report results.
- They should return consistent results from a specified input.
- They should be integrated into our development workflow, so that they are routinely executed. i.e. they need to be automated.

Unit testing is behavioural testing, because we want to test interactions over objects instead of discovering the state of them. In other words we want to test how they behave, based on their interfaces.

How many unit tests should you have? As many as required to cover your critical classes and workflows. (You will NOT get 100% code coverage, despite best intentions).

JUnit Basics

Installing JUnit

To manage our tests, we're going to use **JUnit 5***, a popular testing framework. It can be installed in number of ways: directly from the JUnit home page, or one of the many package managers for your platform.

We'll let Gradle manage JUnit for us as a project dependency. If you look at the `build.gradle` file, you should see these lines, which will force Gradle to manage our test libraries.

```
dependencies {
  // Use the Kotlin test library.
  testImplementation org.jetbrains.kotlin:kotlin-test'

  // Use the Kotlin JUnit integration.
  testImplementation'org.jetbrains.kotlin:kotlin-test-junit'
}
```

* Kent Beck and Eric Gamma invented xUnit, a Smalltalk unit testing framework, while on a flight to OOPSLA in 1997. Over time, it was adapted into nUnit for .NET, CPPUnit for C++ and JUnit for Java.

JUnit Integration

In a Gradle source tree, Tests should be placed under `src/test/kotlin`. It's best practice to have one test class for each class that you want to test.

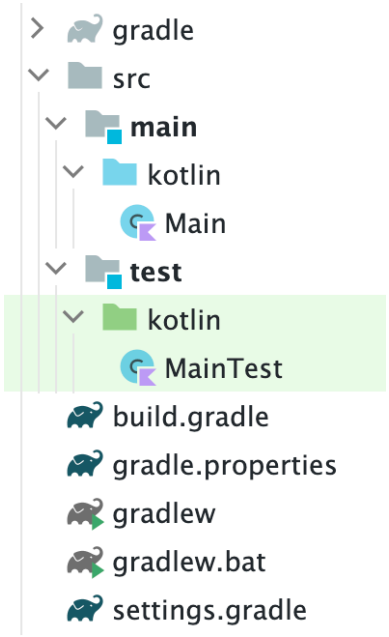
e.g. class Main has a test class MainTest.

Unit tests are executed automatically when you build a project (command-line or from IntelliJ).

```
$ gradle build
BUILD SUCCESSFUL in 928ms
8 actionable tasks: 8 up-to-date // this includes tests
```

We can also execute the gradle "test" task to run them directly.

```
$ gradle test
BUILD SUCCESSFUL in 775ms
3 actionable tasks: 3 up-to-date
```



Gradle build will automatically execute any tests in the test directory structure!

Create a Test Class

1. Create a class under src/main/kotlin.

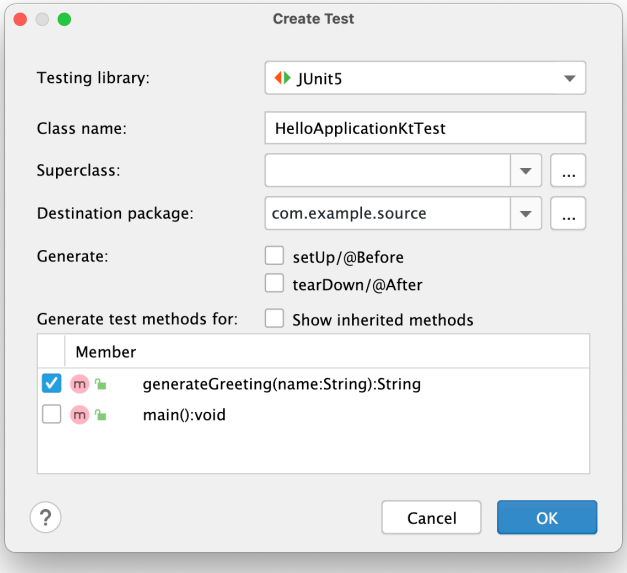
```kotlin
class Sample() {

    fun sum(a: Int, b: Int): Int {
        return a + b
    }
}
```

2. Create a corresponding test class under src/test/kotlin. Add functions, where a function represents a test.

```kotlin
import kotlin.test.Test
import kotlin.test.assertEquals

internal class SampleTest {

    private val testSample: Sample = Sample()

    @Test
    fun testSum() {
        val expected = 42
        assertEquals(expected, testSample.sum(40, 2))
    }
}
```

IntelliJ Tips

1. Press Cmd-N to generate a new test for the selected class.



2. In the test class itself, you can execute a particular test by clicking the Run icon in the gutter.

```
1  package com.example.source
2
3  import org.junit.jupiter.api.Test
4  import org.junit.jupiter.api.Assertions.*
5
6  internal class HelloApplicationKtTest {
7
8      @Test
9      fun generateGreeting() {
10         val expected = "Hello world!"
11         assertEquals(expected, generateGreeting( name: "world"))
12     }
13 }
```

Characteristics of Unit Tests

Here are some general guidelines for writing unit tests.

1. **Tests should be small**. Favour many tests that each check a single thing.
2. **Tests must be independent**.
    - Build up, test and then tear-down any supporting classes.
    - Tests should be able to execute in any order, or in parallel, without assuming any underlying conditions.
    - We should be able to run a single test to focus on a break.
3. **Tests should be consistent**.
    - They should not generate, or rely on side-effects or unpredictable results.

How do you write a unit test?

Every unit test should be a separate function, consisting of the following steps:

1. **Arrange**:
    - Setup the conditions for your test.
    - Initialize variables, load data, setup any dependencies that you might need.
    - Do NOT reuse anything from a different test.
2. **Act**:
    - Execute the functionality that you want to test and capture the results.
3. **Assert**:
    - Check that the actual and expected results match.
    - Use asserts appropriately - see next page.

Example

```
import kotlin.test.Test

class MainTest {
    @Test
    fun saveFile() {
        // FILE 1
        val f1 = "file1.txt"        // arrange
        val file1 = File(f1)

        val status1 = file1.save()  // act

        assert(status1 == FILE.OK)  // assert

        // FILE 2
        val f2 = "file2.txt"        // arrange
        val file2 = File(f2)

        val status2 = file2.save()  // act

        assert(status2 == FILE.OK)  // assert
    }
}
```

Note that we don’t reuse anything from the previous test.

Annotations

The `@Test` annotation tells JUnit that this is a unit test function. The `kotlin.test` package provides annotations to mark test functions, and denote how they are managed:

Annotation	Purpose
@AfterTest	Marks a function to be invoked after each test
@BeforeTest	Marks a function to be invoked before each test
@Ignore	Mark a function to be ignored
@Test	Marks a function as a test

Assertions

We call utility functions to perform assertions of how the function should successfully perform.

Function	Purpose
assertEquals	Provided value matches the actual value
assertNotEquals	The provided and actual values do not match
assertFalse	The given block returns false
assertTrue	The given block returns true

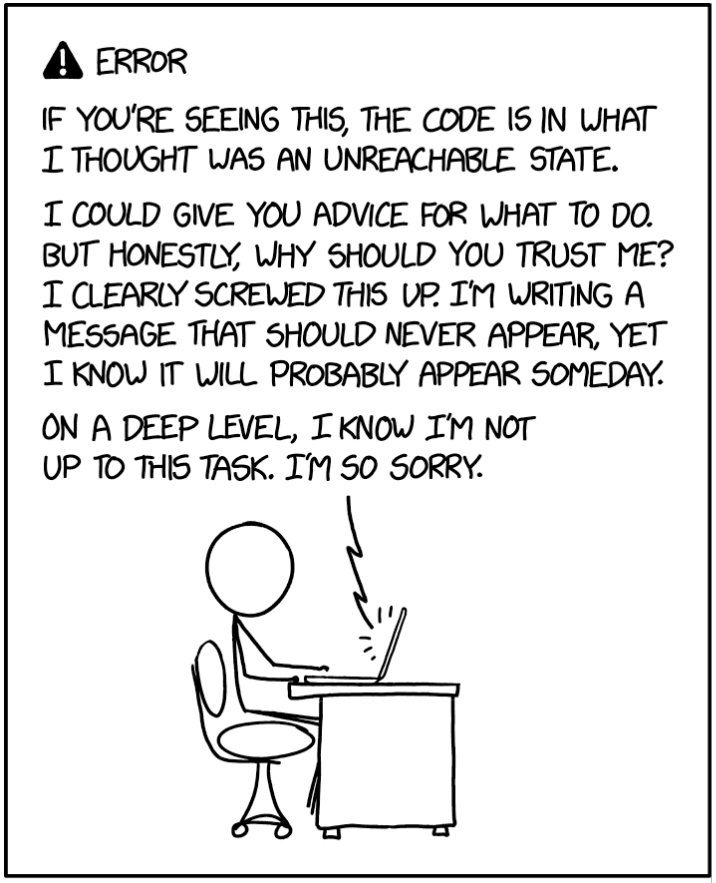
```
class CalcTest {
   @Test
   fun validPlus() {
      val input = arrayOf("1", "+" , "2")
      val results = Calc().calculate(input)}
      assertEquals(3, results)
   }

   @Test
   fun invalidPlus() {
      val input = arrayOf("1", "+", "2")
      val results = Calc().calculate(input)
      assertNotEquals(5, results)
   }

   @Test
   fun insufficientArguments() {
      try {
         val input = arrayOf("1", "+")
         Calc().calculate(input)
      } catch (e:Exception) {
         assertTrue(true)
      }
   }
}
```

Test valid input conditions.
Create a unit test like this for each operation or function.

Test invalid input conditions.
Create a unit test like this for each operation or function to ensure that you handle input errors correctly. Choose representative values (or important outliers)

Special-purpose unit test to check a specific error condition.

```
$ gradle test
BUILD SUCCESSFUL in 760ms
3 actionable tasks: 3 up-to-date
```



NEVER WRITE ERROR MESSAGES TIRED.

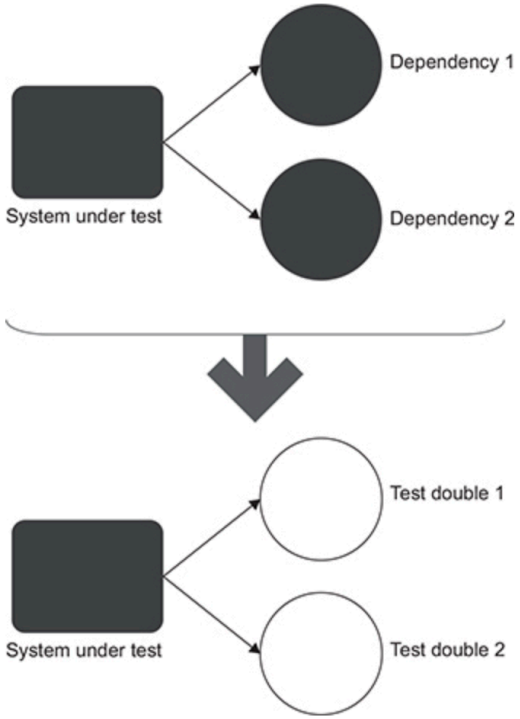
https://xkcd.com/2200/

Mocks

What does “isolation” mean?

- We stated that unit tests should run “in isolation”.
- This means that you should attempt to test just the class in question, independent of other classes.
  - This is difficult due to dependencies: classes often rely on the behaviour of other classes to work correctly.
- To accomplish this, we often create **test doubles** — classes that are meant to look like a dependent class, but that don’t actually implement all of the underlying behaviour.
  - This lets us swap in these “fake” classes for testing, to simplify testing.

There are five principal kinds of test doubles: **Dummies, Fakes, Stubs, Spies** and **Mocks.**



Mocks

Martin Fowler describes mocks as “objects pre-programmed with expectations which form a specification of the calls they are expected to receive.”

A mock is a fake object that holds the expected behaviour of a real object but without any genuine implementation. For example, we can have a mocked File System that would report a file as saved, but would not actually modify the underlying file system.

You can fairly easily create these mock classes yourself for code domain objects.

Several libraries have also been established to help create mocks of objects. Mockito is one of the most famous, and it can be complemented with Mockito-Kotlin. Here’s an example of a mock File that reports a path, but doesn’t actually do anything else.

```
private val mockedFile: File {
  return mock { on { absolutePath} doReturn "/random"}
}
```

Code Coverage

How many tests do I need?

“Tests shouldn’t verify units of code. Rather, they should verify units of behaviour: something that is meaningful for the problem domain and, ideally, something that a business person can recognize as useful. The number of classes it takes to implement such a unit of behaviour is irrelevant. ” — Khorikov (2020)

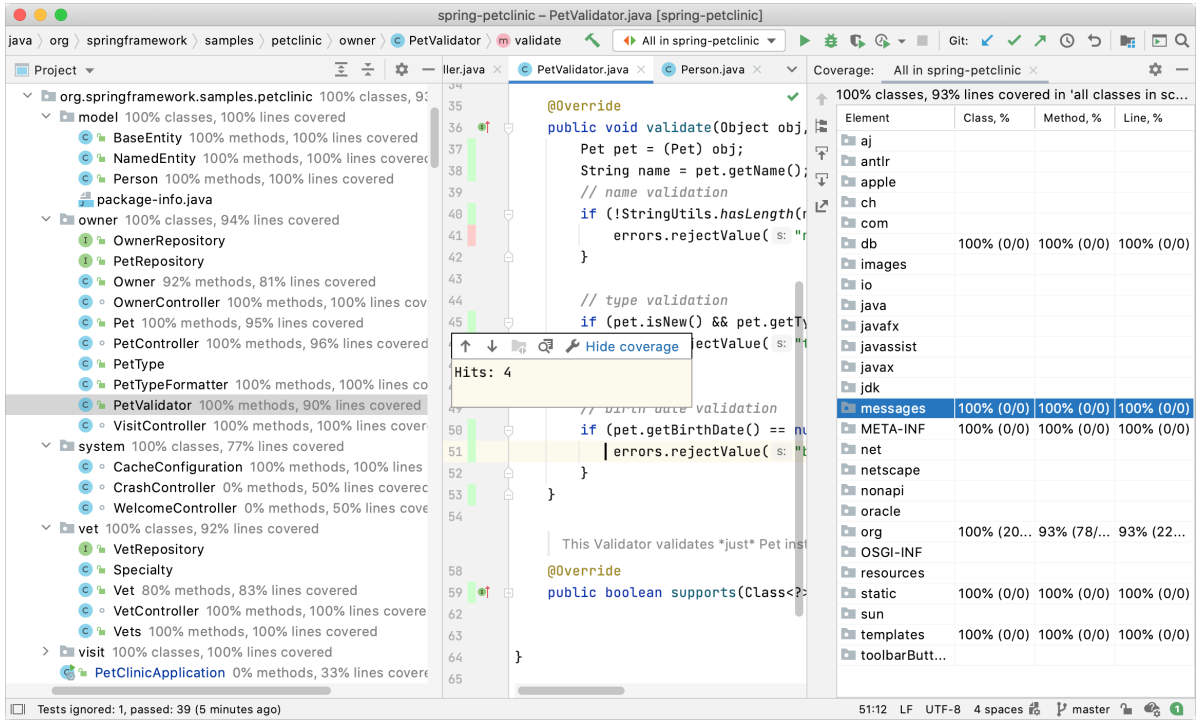
Code coverage is a metric comparing the number of lines of code with the number of lines of code that have unit tests covering their execution. In other words, what “percentage” of your code is tested?

This is a misleading statistic at the best of times (we can easily contrive cases where code will never be executed or tested).

TDD would suggest that you should have 100% unit test coverage but this is impractical and not that valuable. You should focus instead on covering key functionality. e.g. domain objects, critical paths of your source code.

IntelliJ Coverage

- One recommendation is to look at the coverage tools in IntelliJ, which will tell you how your code is being executed, as well as what parts of your code are covered by unit tests.
- Use this to determine which parts of the code should be tested more completely.



https://www.jetbrains.com/help/idea/code-coverage.html

https://www.jetbrains.com/help/idea/running-test-with-coverage.html